

University Politehnica of Bucharest  
Faculty of Automatic Control and Computers  
Computer Science and Engineering Department

Diploma Thesis

# **Mobile Gateway for Wireless Sensor Networks utilizing drones**

by

**Ioan Deaconu**

Supervisor: Prof. Dr. Ing. Nicolae Țăpuș  
Supervisor: As. Drd. Ing. Andrei Voinescu

Bucharest, September 2014

---

# Contents

---

<b>Contents</b>	<b>i</b>
<b>1 Introduction</b>	<b>3</b>
<b>2 Related Work</b>	<b>4</b>
<b>3 Hardware Platform</b>	<b>5</b>
3.1 The Parrot AR.Drone 2.0 . . . . .	6
3.2 The Sparrow Dongle . . . . .	8
3.3 The SparrowV32 . . . . .	9
<b>4 Software Environment: Contiki Operating System</b>	<b>10</b>
4.1 The Data Collecting Module . . . . .	12
4.1.1 Modules intercommunication . . . . .	12
4.1.2 Failure proof . . . . .	13
4.2 The Communication Module . . . . .	15
4.2.1 Socket with connection reset . . . . .	15
4.2.2 JSON Encoding of Data . . . . .	18
<b>Bibliography</b>	<b>20</b>
<b>A Contiki API</b>	<b>22</b>
A.1 Process macros . . . . .	22
A.2 uIP functions . . . . .	22
<b>B Node capabilities</b>	<b>24</b>
<b>List of Figures</b>	<b>25</b>
<b>List of Tables</b>	<b>26</b>
<b>Listings</b>	<b>27</b>

---

# Abstract

---

**Keywords** Wireless Sensor Networks, task, scheduling, graph cuts

---

# Acknowledgements

---

## **Chapter 1**

---

# **Introduction**

---

Thesis Intro - no more than 3 pages.

## Chapter 2

---

# Related Work

---

Related work for task scheduling

## Chapter 3

---

# Hardware Platform

---

In this chapter we will present the hardware platforms uses in order  
chestu ..... help andrei

### 3.1 The Parrot AR.Drone 2.0

Parrot AR.Drone is a wifi radio controlled flying quadcopter built by the French company Parrot. The original drone was released in 2010 and in 2012 it was replaced by version 2.0. Since the launch of the original AR.Drone, more the half a million units have been sold, making it one of the, if not, the most popular drone on the market.

The reason of its success is not entirely due to the relatively low price of around 300\$ but also because it is very easy to learn how to control the drone and also because of the usb port that accomodate any device using that interface and the linux operating system



Figure 3.1: *The arrot AR.Drone 2.0*

Because of those reasons, the Drone has a number of aftermarket modules that can be attached to it like the Flight Recorder GPS Module. This module has a built in storage of 4GB for video recording purposes and a built in GPS receiver. This allows the drone to follow a predetermined path of waypoints and to return back from where it took off automatically, all within the limit of the Wi-Fi connection with the control device.

The arrot AR.Drone 2.0 specifications are :

- 1GHz 32 bit ARM Cortex A8 processor with 800MHz video DSP TMS320DMC64x
- Linux 2.6.32
- 1Gbit DDR2 RAM at 200MHz
- USB 2.0 high speed for extensions



- Wi-Fi b,g,n
- 3 axis gyroscope 2000/second precision
- 3 axis accelerometer  $\pm 50\text{mg}$  precision
- 3 axis magnetometer 6 precision
- Pressure sensor  $\pm 10\text{ Pa}$  precision
- Ultrasound sensors for ground altitude measurement
- 60 fps vertical QVGA camera for ground speed measurement
- 30 fps 720p front mounted camera

## 3.2 The Sparrow Dongle

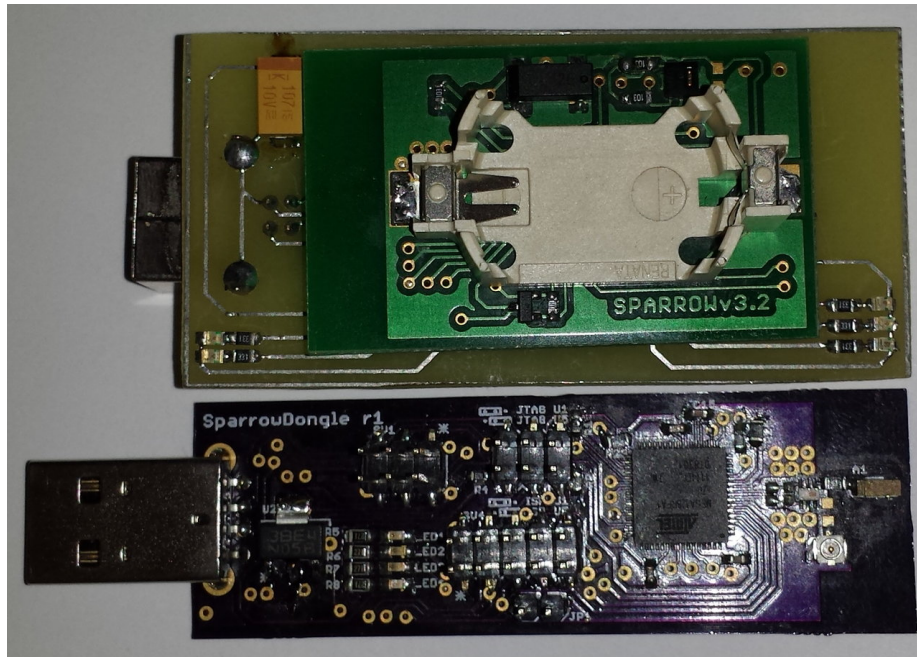


Figure 3.2: *The Sparrow Dongle next to the SparrowV32*

### 3.3 The SparrowV32

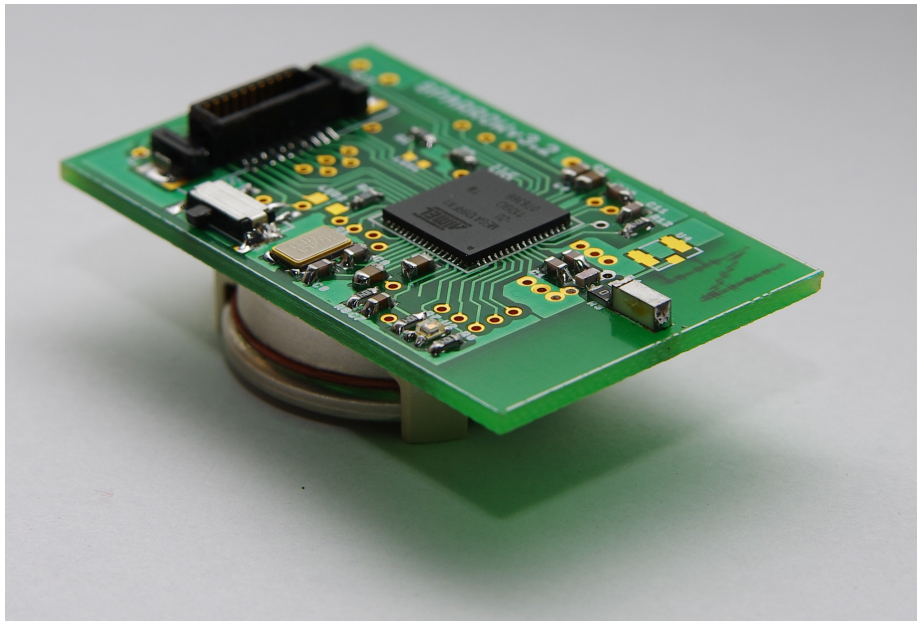


Figure 3.3: *The SparrowV32*

## Chapter 4

---

# Software Environment: Contiki Operating System

---

software intro ....

The software is composed on diferent interlocking (trebuie modificat ) modules running on diferent devices and operating systems.

The modules are written mainly in java and c.

again ... andrei ?

The code can be compiled to display debug informations to the console or to supress them entirely. Also, in certain parts of the modules, a wait action is need in order to wait for an action to be executed. Now the delay is set at 100 ms, but it can be easily modified to any desired value.

---

Listing 4.1: Data Collection use of mutex

---

```
/* activates/deactivates printf debug information*/  
#define DEBUG_ON 0  
/* delay yime in microseconds*/  
#define DELAY_US 100000  
#define DEBUG_PRINT(a...) { if(DEBUG_ON) printf(a); }
```

---

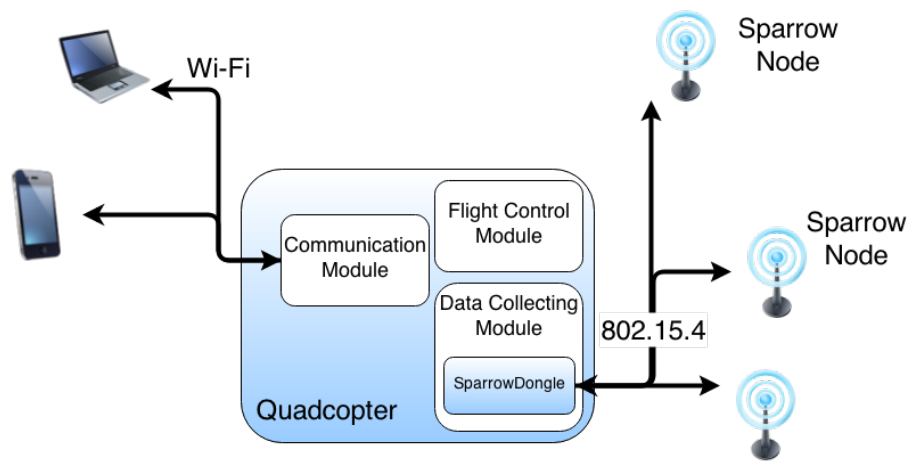


Figure 4.1: *Modules and connections between them and devices*

## 4.1 The Data Collecting Module

The module saves the collected data into the drones internal memory and passes the data in order to gather certain informations like number of nodes currently connected to the Dongle, the signal strength, if the Dongle is connected etc. This informations are passed to the communication module to provide to the user realtime feedback.

### 4.1.1 Modules intercommunication

The memory area in which the informations sent to the user are saved is shared between this module and the communication module. Basically, the way this two modules interact with each other can be compared to the consumer - producer problem, where the Data Collecting Module can be associated with the producer side and the Communication Module with the consumer side.

The main problem consists of deadlocks and data starvation. This is prevented with the use of one mutex that allows only one thread at a time to modify the informations.

---

Listing 4.2: Data Collection use of mutex

---

```
pthread_mutex_lock(&data_lock);
add_node_data(get_current_timestamp(), read_data + 7);
pthread_mutex_unlock(&data_lock);
```

---

The mutex is used similarly in the Communication Module when it consumes the information.

---

Listing 4.3: Data Collection use of mutex

---

```
void add_node_data(long long time_stamp , char *p) {
    int i;
    int id = get_hex(p,2);

    /* the power of the signal calculated in dB */
    int power = -90 + 3* (get_hex(p + 64,2)-1);

    /* creating a file with unique name */
    DEBUG_PRINT("node id %i %i\n",id,power);
    char file_name[100];
    sprintf(file_name, "/node_logs/%lli_%i",file_timestamp,id);

    /* saving the new data at the end of the file */
    FILE *fptr = fopen(file_name,"a");
    fprintf(fptr,"%s",p);
```

```

fclose(fp_ptr);

/* searching for previous connection of the same node */
for(i = 0 ; i < node_nr; i++) {
    if(data[i].id == id) {

        /* timestamp update - node is still reachable and sending data */
        DEBUG_PRINT("data update\n");
        data[i].time_stamp = time_stamp;
        data[node_nr].power = power;
        return;
    }
}

/* new node found by the drone */
DEBUG_PRINT("new node\n");
data[node_nr].id = id;
data[node_nr].time_stamp = time_stamp;
data[node_nr].power = power;
node_nr++;
}

```

---

#### 4.1.2 Failure proof

Because the Dongle is connected to an USB port on a machine that has a lot of vibrations, it might disconnect / reconnect for a very short period of time, so this module has been designed with multiple USB disconnects and reconnects without the need to reset the Drone. This information is vital, because you can check if the Dongle is still connected to the drone without the need to inspect it visually or to connect to a debug terminal.

---

Listing 4.4: Data Collection use of mutex

---

```

while(1) {
    DEBUG_PRINT("attempting connection to Sparrow Dongle\n");
    FILE *ptr = NULL;
    while(ptr == NULL) {
        usleep(DELAY_US);
        ptr = fopen("/dev/ttyACM0", "r");
    }

    dongle_connected = 1;
    DEBUG_PRINT("Sparrow Dongle connected\n");
}

```

```
char read_data[100];

while(fgets(read_data,100,ptr) != NULL)
{
    if(check_message_format(read_data)) {
        pthread_mutex_timedlock(&data_lock,&lock_timeout);
        add_node_data(get_current_timestamp(),read_data + 7);
        pthread_mutex_unlock(&data_lock);
    }
}
dongle_connected = 0;
fclose(ptr);
DEBUG_PRINT("Sparrow Dongle Disconnected\n");
}
```

---



## 4.2 The Communication Module

All the information gathered by the Data Collecting Module would be useless if it cannot be accessed easily.

This module, as the name suggests, handles the communication of this crucial information back to the user.

Being a different module, with different attributions than the Data Collecting Module, it has an entire process dedicated to it for 3 important reasons:

- 1. This approach of a module with its one process allows the modules to run independently of each other;
- 2. The Data Collecting Module can collect the data from the Dongle as soon as it has a new one available;
- 3. If the Communication Module stops working, the Data Collecting Module continues to save the new informations received.

### 4.2.1 Socket with connection reset

The communication is done through socket connections listening on port 8888. It accepts only one connection at a time.

If a connected client decides to disconnect before or while a write is performed, a SIGPIPE error signal will be thrown, stopping all the modules. This is prevented by ignoring the signal, forcing the write action to return a EPIPE, an exiting the this function.

The main process will callback the `accept_socket_connection` to reestablish a new connection.

Listing 4.5: Data Collection use of mutex

---

```
// Ignore the SIGPIPE error signal
signal(SIGPIPE, SIG_IGN);

struct sockaddr_in server;

//Create socket
socket_desc = socket(AF_INET , SOCK_STREAM , 0);
if (socket_desc == -1)
{
    DEBUG_PRINT("Could not create socket\n");
    return;
}
DEBUG_PRINT("Socket created\n");
```

```

//Prepare the sockaddr_in structure
server.sin_family = AF_INET;
server.sin_addr.s_addr = INADDR_ANY;
server.sin_port = htons( 8888 );

//Bind
if( bind(socket_desc,(struct sockaddr *)&server , sizeof(server)) < 0)
{
    //print the error message
    DEBUG_PRINT("bind failed. Error\n");
    return ;
}
DEBUG_PRINT("bind done\n");

//Listen
listen(socket_desc , 3);

while(1){

    accept_socket_connection();

}

```

---

This function will listen for a new connection. Once a connection is established, it will send information once every DELAY\_US microseconds. The program was configured and tested with a 100 ms wait period that leads to a ten times per second information update.

This delay is required because:

- If data is send to often , the socket might be flooded and stop sending the data
- This will create an big and useless processor use both for the drone and the controlling device.

---

Listing 4.6: Data Collection use of mutex

---

```

void accept_socket_connection() {

    struct sockaddr client;
    char *client_message ;
    int  client_sock , c , read_size;

```

```

//Accept and incoming connection
DEBUG_PRINT("Waiting for incoming connections...\n");
c = sizeof(struct sockaddr_in);

//accept connection from an incoming client
client_sock = accept(socket_desc, (struct sockaddr *)&client, (socklen_t *)&client_sock);
if (client_sock < 0)
{
    DEBUG_PRINT("accept failed\n");
    return ;
}
DEBUG_PRINT("Connection accepted\n");

//Receive a message from client
while(1)
{
    // sleep in order to avoid flooding the socket
    usleep(DELAY_US);

    client_message = json_encode();
    // error writing - client disconnected
    if (write(client_sock , client_message , strlen(client_message)) != strlen(client_message))
    {
        free(client_message);
        DEBUG_PRINT("Client disconnected \n" );
        // reseting listening procces
        return;
    }
    free(client_message);
}

if(read_size == 0)
{
    DEBUG_PRINT("Client disconnected\n");
    fflush(stdout);
}
else if(read_size == -1)
{
    DEBUG_PRINT("recv failed\n");
}
}

```

---

### 4.2.2 JSON Encoding of Data

The data sent is encoded in JSON format, because it is very easy to encode and all devices can decode it.

The informations ecoded are

- Dongle conection status
- An array containing node informations
  - Node unique ID
  - Last conection time of the node to Dongle
  - The power of the received signal

---

Listing 4.7: Data Collection use of mutex

---

```
char * json_encode(){
    int i,msg_index = 0;

    char *client_message = (char*)malloc(3000 * sizeof(char));
    client_message[0]='\0';

    pthread_mutex_timedlock(&data_lock,&lock_timeout);
    current_timestamp = get_current_timestamp();
    for(i = 0; i < node_nr;i++) {
        if(current_timestamp - data[i].time_stamp < delta)
        {
            client_message[0]++;
        } else {
            delete_node_data(i);
            i--;
        }
    }

    int local_node_nr = node_nr;
    struct node_data local_data[node_nr];
    memcpy(local_data,data,node_nr * sizeof(struct node_data));
    pthread_mutex_unlock(&data_lock);

    msg_index += sprintf(client_message+msg_index, "{ \"dongle_connected\"="
```

```
current_timestamp = get_current_timestamp();
for(i = 0; i < local_node_nr; i++) {
    if(i > 0 ) {
        msg_index += sprintf(client_message+msg_index, ",");
    }
    msg_index += sprintf(client_message+msg_index, "{\"node_id\"=%i,\"1
}

msg_index += sprintf(client_message+msg_index, "]]\n");

return client_message;
}
```

---

---

# Bibliography

---

- [Atm] Atmel. Rzraven hardware user's guide. [cited at p. -]
- [Cal04] Edgar H. Callaway. *Wireless Sensor Networks: Architectures and Protocols*. CRC Press, 2004. [cited at p. -]
- [CK03] C. Chong and S.P. Kumar. In *Sensor networks: Evolution, opportunities, and challenges*, 2003. [cited at p. -]
- [DPdR<sup>+</sup>05] Flvia Coimbra Delicato, Fbio Protti, Jos Ferreira de Rezende, Luiz F. Rust da Costa Carmo, and Luci Pirmez. Application-driven node management in multihop wireless sensor networks. In Pascal Lorenz and Petre Dini, editors, *ICN (1)*, volume 3420 of *Lecture Notes in Computer Science*, pages 569–576. Springer, 2005. [cited at p. -]
- [ea07] Roberto Verdone et al. *Wireless Sensor and Actuator Networks: Technologies, Analysis and Design*. Academic Press, 1st edition, 2007. [cited at p. -]
- [GH88] Olivier Goldschmidt and Dorit S. Hochbaum. Polynomial algorithm for the k-cut problem. 1988. [cited at p. -]
- [HJ05] Tarek Hagraas and Jan Janecek. A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems. *Parallel Computing*, 31(7):653–670, 2005. [cited at p. -]
- [Jac06] Brian Jacokes. Lecture notes on multiway cuts and k-cuts, July 2006. [cited at p. -]
- [KD06] Snehal Kamalapur and Neeta Deshpande. Efficient cpu scheduling: A genetic algorithm based approach. *Ad Hoc and Ubiquitous Computing*, pages 206 – 207, December 2006. [cited at p. -]
- [Li08] Xiang-Yang Li. *Wireless Ad Hoc and Sensors Networks: Theory and Applications*. Cambridge University Press, 2008. [cited at p. -]
- [NGT99] Andrew Goldberg Nec, Andrew V. Goldberg, and Kostas Tsioutsoulklis. Cut tree algorithms. In *In Symposium on Discrete Algorithms*, pages 376–385, 1999. [cited at p. -]

- [PB05] C. Prehofer and C. Bettstetter. Self-organization in communication networks: principles and design paradigms. *IEEE Communications Magazine*, 43(7):78–85, July 2005. [cited at p. -]
- [TEÖ07] Yuan Tian, Eylem Ekici, and Füsün Özgüner. Energy-constrained task mapping and scheduling in wireless sensor networks. 2007. [cited at p. -]
- [YJK07] William M Iversen Yunseop (James) Kim, Robert G Evans. The future of intelligent agriculture. *Resource*, October 2007. [cited at p. -]
- [zig] Zigbit wireless modules datasheet. [cited at p. -]

## Appendix A

---

# Contiki API

---

### A.1 Process macros

- `PROCESS_THREAD (name, ev, data )` - Define the body of a process. This macro is used to define the body (protothread) of a process. The process is called whenever an event occurs in the system, A process always starts with the `PROCESS_BEGIN()` macro and end with the `PROCESS_END()` macro.
- `PROCESS_BEGIN ()` - Define the beginning of a process.
- `PROCESS_END ()` - Define the end of a process.
- `PROCESS_YIELD ()` - Yields the currently running process
- `PROCESS_WAIT_EVENT_UNTIL (c)` - Wait for an event to be posted to the process, with an extra condition. This macro is very similar to `PROCESS_WAIT_EVENT()` in that it blocks the currently running process until the process receives an event. But `PROCESS_WAIT_EVENT_UNTIL()` takes an extra condition which must be true for the process to continue.
- `PROCESS_PAUSE` - Yield the process for a short while. This macro yields the currently running process for a short while, thus letting other processes run before the process continues.

### A.2 uIP functions

- `PSOCK_INIT (psock, buffer, buffersize)` - Initializes a proto-socket. This macro initializes a protosocket and must be called before the pro-



tosocket is used. The initialization also specifies the input buffer for the protosocket.

- `PSOCK_SEND (psock, data, datalen)` - Send data. This macro sends data over a protosocket. The protosocket protothread blocks until all data has been sent and is known to have been received by the remote end of the TCP connection.
- `PSOCK_READBUF (psock)` - Read data until the buffer is full. This macro will block waiting for data and read the data into the input buffer specified with the call to `PSOCK_INIT()`. Data is read until the buffer is full..
- `CCIF process_event_t tcpip_event` - The uIP event. This event is posted to a process whenever a uIP event has occurred.
- `CCIF void tcp_listen (u16_t port)` - Open a TCP port. This function opens a TCP port for listening. When a TCP connection request occurs for the port, the process will be sent a `tcpip_event` with the new connection request.
- `struct uip_conn *tcp_connect(uipaddr_t *ripaddr, u16 port, void *appstate)` - This function opens a TCP connection to the specified port at the host specified with an IP address. Additionally, an opaque pointer can be attached to the connection. This pointer will be sent together with uIP events to the process.
- `uip_connected()` - Has the connection just been connected?
- `uip_closed()` - Has the connection been closed by the other end?
- `uip_aborted()` - Has the connection been aborted by the other end?
- `uip_timedout()` - Has the connection timed out?
- `uip_newdata()` - Is new incoming data available?
- `uip_close()` - Close the current connection.

## Appendix B

---

# Node capabilities

---

Task	AVR Raven™	Sparrow	Sparrow Power
Temperature sensing	✓	✓	
Humidity sensing		✓	
Voltage & Current sensing			✓
Event detection	✓	✓	✓
Alarm beep	✓		
LED signal	✓	✓	

Table B.1: Node capabilities

---

# List of Figures

---

3.1	<i>The arrot AR.Drone 2.0 . . . . .</i>	6
3.2	<i>The Sparrow Dongle next to the SparrowV32 . . . . .</i>	8
3.3	<i>The SparrowV32 . . . . .</i>	9
4.1	<i>Modules and connections between them and devices . . . . .</i>	11

---

# List of Tables

---

B.1 Node capabilities . . . . .	24
---------------------------------	----

---

# Listings

---

4.1	Data Collection use of mutex . . . . .	10
4.2	Data Collection use of mutex . . . . .	12
4.3	Data Collection use of mutex . . . . .	12
4.4	Data Collection use of mutex . . . . .	13
4.5	Data Collection use of mutex . . . . .	15
4.6	Data Collection use of mutex . . . . .	16
4.7	Data Collection use of mutex . . . . .	18